Local Concurrency Detection in Business Process Event Logs

ABEL ARMAS-CERVANTES, Queensland University of Technology, Australia
MARLON DUMAS, University of Tartu, Estonia and Queensland University of Technology, Australia
MARCELLO LA ROSA, Queensland University of Technology, Australia
ABDERRAHMANE MAARADJI, Queensland University of Technology, Australia

Detecting concurrency relations between events is a fundamental primitive underpinning a range of process mining techniques. Existing approaches to this problem identify concurrency relations at the level of event types under a global interpretation. If two event types are declared to be concurrent, every occurrence of one event type is deemed to be concurrent to one occurrence of the other. In practice, this interpretation is too coarse-grained and leads to over-generalization. This paper proposes a finer-grained approach, whereby two event types may be deemed to be in a concurrency relation relative to one state of the process, but not relative to other states. In other words, the detected concurrency relation holds locally, relative to a set of states. Experimental results both with artificial and real-life logs show that the proposed local concurrency detection approach improves the accuracy of existing concurrency detection techniques.

Categories and Subject Descriptors: Applied computing [**Enterprise computing**]: Business process management; Information systems [**Information systems applications**]: Data mining

Additional Key Words and Phrases: Process mining, concurrency oracle, event structure

ACM Reference Format:
Armas-Cervantes, A., Dumas., M., La Rosa, M., Maaradji, A. 2017. Local Concurrency Detection in Business Process Event Logs. ACM V, N, Article A (January YYYY), 18 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Process mining is a body of techniques that help analysts understand business processes based on their event logs. In this context, an event log is a set of traces, each consisting of a sequence of events with associated attributes. Each event is an instance of an event type, corresponding to an activity or business-relevant event in the process. For example, an event log of an order-to-cash process may include event types such as "Goods shipped" and "Payment collected". An event of type "Payment collected" may include additional attributes such as the confirmation number and the amount. Given an event log, process mining tools can extract a process model (*automated process discovery*), check the conformance of a given process model against the log (*conformance checking*), compare two event logs (*log delta analysis*), or detect changes in the execution of a process over time (*drift detection*), among other analysis operations.

A number of process mining techniques rely on the identification of behavioral relations between pairs of events, most notably causality relations (the occurrence of an event entails the subsequent occurrence of another one), conflict relations (the occurrence of an event excludes the occurrence of another event) and interleaved concurrency relations (two events co-occur in any order). A key challenge in this context is

Authors' addresses: A. Armas-Cervantes, M. La Rosa and A. Maaradji, Queensland University of Technology, GPO Box 2434, Brisbane, QLD 4001, Australia; M. Dumas, University of Tartu, J. Liivi 2, Tartu 50409, Estonia.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

how to accurately distinguish between concurrency and causality relations, particularly in the presence of repetitive behavior (loops). Several automated process discovery [van Dongen et al. 2012; Ponce-de León et al. 2015], conformance checking [Lu et al. 2014; García-Bañuelos et al. 2015], delta analysis techniques [van Beest et al. 2015] and drift detection [Maaradji et al. 2015] techniques take as input a *concurrency oracle*, i.e. a black-box boolean function that asserts whether a given pair of events are concurrent or not, to transform the traces in the log (totally ordered set of events) into partially ordered sets of events.

Existing approaches to implement concurrency oracles – e.g. the ones embedded in the α process discovery algorithm and its variants [van der Aalst et al. 2004; de Medeiros et al. 2004; Wen et al. 2007; Li et al. 2007] – detect *global concurrency relations*, at the level of pairs of event types. The semantics of a global concurrency relation between two event types is that an instance of the first type must be either followed or preceded by an instance of the second type regardless of where in the log these instances occur. In practice, this property does not always hold. For example, consider a log recording the executions of the process for plan lodgement and document registration in two different Australian states, South and Western Australia, whose model is shown in Figure 1.[1] A global concurrency oracle would assert that event types "Update register" and "Update DCDB", and event types "Approve plan" and "Update register", among others, are concurrent. However, "Approve plan" and "Update register" are concurrent only in the case of Western Australia (i.e. when the WA path after the decision point is taken), while "Update register" and "Update DCDB" are never concurrent. This approximation then affects the precision of the process mining techniques. For example in the context of automated process discovery, the result is likely to be a model where activities "Update register" and "Update DCDB"' can always occur in any order regardless of the state.

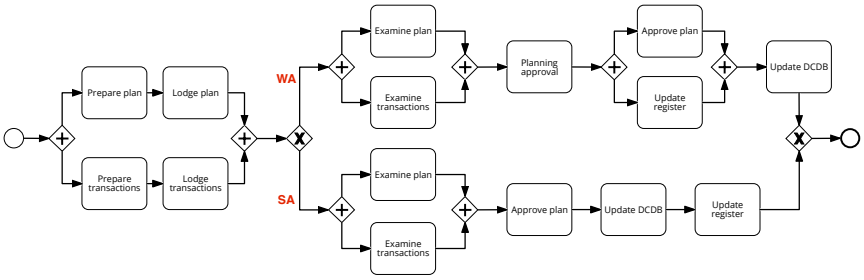


Fig. 1: Process model for plan lodgement and document registration in Western Australia (WA) and South Australia (SA).

This paper advocates an alternative *local concurrency* detection approach whereby a concurrency relation between two event types is scoped to certain execution states of the process. The main contribution is an approach that turns any global concurrency oracle into a scoped (local) one. The key idea is to construct a state transition graph from the event log and to traverse this graph in order to discover concurrency relations (using an existing concurrency oracle) in-between pairs of states. The accuracy of the proposed local concurrency detection approach is compared against the α global concurrency oracle and inductive miner [Leemans et al. 2013], a well-known process discovery algorithm, based on a synthetic testbed comprising a range of combinations of control-flow structures, as well as seven real-life logs. We also show on a real-life process model that the proposed concurrency oracle can improve the accuracy of an existing business process drift detection technique.

[1] This model forms part of a collection of real-life process models for handling land development applications in Australia.

The rest of the paper is structured as follows. Section 2 discusses existing approaches to construct concurrency oracles and their limitations. Section 3 introduces the proposed approach, while Sections 4-6 present its experimental evaluation. Finally, Section 7 summarizes the results and outlines future work directions.

2. RELATED WORK

Several techniques have approached the problem of discovering behavioral relations (in particular concurrency relations) between pairs of event types. For instance, [Cook and Wolf 1998] outlines a technique based on statistical measures to discover event-based models that capture the concurrent execution of events types. This latter technique is however highly dependent on the quality of the log. In particular, it assumes that concurrency is embedded in blocks that have a single split and a single join event, and that the order of occurrence of the concurrent event types is uniformly distributed. The α-algorithm [van der Aalst et al. 2004] and its variants [de Medeiros et al. 2004; Wen et al. 2007; Li et al. 2007] detect concurrency relations (among other behavioral relations) between event types. The α-algorithm itself declares a pair of event types to be concurrent if one immediately precedes the other and vice-versa. Both mentioned approaches, α-algorithm and that presented in [Cook and Wolf 1998], lead to global concurrency oracles, which as stated before, disregard the context where the events occur, thus leading to false positives.

Extensions of the α-algorithm such as $\alpha+$ [de Medeiros et al. 2004] are designed to prevent the α-algorithm from confusing concurrency with (short) loops and other limitations, however, they still suffer from the limitation of being global.

In [van Dongen and van der Aalst 2004], the relations computed by the α-algorithm (referred to as α relations hereinafter) are used as a concurrency oracle to construct a partially ordered run (therein called an *instance graph*) from each trace in an event log. The resulting set of runs can be used to synthesize a process model (e.g. a Workflow net) [van Dongen et al. 2012]. This latter approach however inherits the limitations of the α-algorithm as a method for constructing concurrency oracles. A more recent approach to construct partially ordered runs [Diamantini et al. 2016] from traces addresses the issue of discovering concurrency in the presence of infrequent event types. The starting point is still a process discovery algorithm that incorporates a global concurrency oracle. Traces are turned into partially ordered runs based on the global oracle and then adjusted to take into account infrequent event types.

In the context of process model synthesis, some techniques require additional data for the computation of concurrency relations. For example, the approach presented in [Leemans et al. 2015] requires that a log contains the start and end timestamps of every event, and then a pair of events are concurrent if they overlap in time. However, the information about the start and end timestamps of an event is not always available in the event logs.

A technique to discover scoped concurrency relations between events is proposed in [Mokhov and Carmona 2015]. Given an event log, this technique produces a conditional partial order graph [Mokhov and Yakovlev 2010]. In this graph, a concurrency relation is scoped by means of (data) conditions, i.e. the concurrency relation only holds when the condition evaluates to true at a given point in the process. A condition is determined by the execution of an event, e.g. a and b are concurrent if c is executed, noting that c does not necessarily need to occur before a and b. However, this technique makes the highly restrictive assumption that there are no two events of the same type in the same trace, since when a duplicate event is found in a trace, the trace is split and the two sub-traces are treated as two different traces, leading to the possibility of identifying concurrency across these traces.

3. DISCOVERING LOCAL CONCURRENCY

An overview of the proposed approach is given in Fig. 2. The first step consists in abstracting the execution state information represented in an event log. To this end a transition is constructed, where each trace in the log is a path from the initial state to a final state in this graph. The step requires an equivalence relation ($\equiv$), which is used to find similar execution states in the process. Given the transition graph, the second step is to identify concurrency relations that hold between two states. This step takes as input a global concurrency oracle ($\mathcal{C}$) and a validation function ($\mathcal{F}$). $\mathcal{C}$ can be for example the α-algorithm or other oracles mentioned in the previous section. Since a given concurrency relation may hold to various degrees in multiple scopes, including pairs of scopes such that one contains the other, it is necessary to select its most suitable scope. To this end, $\mathcal{F}$ is used to assess the likelihood of the computed concurrency relations. Note that the equivalence relation, the global concurrency oracle and the validation function can be customized; however, in this paper we present a single configuration of these three elements.

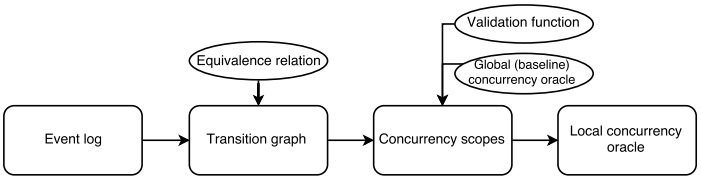


Fig. 2: Proposed approach.

This section starts by presenting the construction of the transition graph and then the computation of the scopes.

3.1. Transition graph of an event log

The first step of our approach consists of constructing a transition graph representing the behavior captured in an event log, such that every transition in the graph represents the execution of an event (or event type) and every state represents the occurrence of some events (or event types). In fact, this type of representation has been widely used in the context of process mining. For instance, [van der Aalst et al. 2008] presents several strategies for the construction of a transition graph that can be modified to vary the degree of generalization. However, the approach presented in [van der Aalst et al. 2008] aims at constructing a transition graph from which a model is synthesized using theory of regions, and considers steps where transitions are added or removed from the transition graph. Different from [van der Aalst et al. 2008], our approach neither adds nor removes transitions in such graph.

Before presenting the construction of a transition graph, we define some notations on sequences, traces and event logs.

Definition 3.1 (*Sequences and subsequences*). Let A be a set of elements. A sequence σ over A is denoted by $\sigma = \langle a_1\ a_2\ a_3\ \dots\ a_n\rangle \in A^*$, $n \in \mathbb{N}$; whereas an *empty* sequence is denoted by ϵ. The length of a sequence σ is number of elements it contains and is denoted by $|\sigma|$, e.g., $|\langle a_1\ a_2\ a_3\rangle| = 3$.

A *prefix subsequence* of length m of a sequence $\sigma = \langle a_1\ a_2\ a_3\ \dots\ a_n\rangle$ is another sequence composed by the first m elements, and it is shorthanded as $\sigma_{[1\dots m]} = \langle a_1\ a_2\ a_3\ \dots\ a_m\rangle$, $0 \leq m \leq n$. The set of all prefix subsequences of $\sigma = \langle a_1\ a_2\ a_3\ \dots\ a_n\rangle$ is represented as $\phi(\sigma) = \{\sigma_{[1\dots k]} \mid 0 \leq k \leq n\}$. An element e_x is an extension of a prefix subsequence $\sigma' = \langle a_1\ a_2\ \dots a_l\rangle$ of σ, denoted as $\sigma' \oplus e_x$, if $\langle a_1\ a_2\ \dots a_l\ e_x\rangle \in \phi(\sigma)$.

A *suffix subsequence* of length m of $\sigma = \langle a_1\ a_2\ a_3\ \dots\ a_n\rangle$ is the sequence composed by the last m elements of σ, and it is denoted as $\sigma_{[m,n]} = \langle a_{n-m}\ \dots a_{n-1}\ a_n\rangle$, $0 \leq m \leq n$.

An event log is a set of traces,[2] each describing a sequence of events over a set of activities Λ.

Definition 3.2 (*Trace, Event log*). Given a set of activities Λ, let E be a set of events and $\lambda : E \to \Lambda$ be a labelling function. A *trace* is a sequence of events $\sigma = \langle \lambda(e_1), \lambda(e_2), \ldots, \lambda(e_n) \rangle$ for $e_i \in E$, where $1 \leq i \leq n$. Finally, an *event log* is a set of traces and it is denoted as $\mathcal{L}$.

Given that a trace σ can contain several occurrences of the same activity, we adopt the following convention: every event is unique within a trace and is represented by a label and an index. The label of the event is the name of the activity it represents and the index is the occurrence number of the activity in σ. For instance, the trace $\langle a\ b\ b \rangle$ is composed by one occurrence of activity a and two occurrences of activity b, thus the events in the trace would be $\langle a_1\ b_1\ b_2 \rangle$. By the abuse of notation, we refer to any generic event as e_i where $i \in \mathbb{N}$. We say that two events e_i, e_j in different traces are equivalent, denoted as $e_i \sim e_j$, if they are instances of the same activity and represent the same number of occurrence within their traces, i.e., $\lambda(e_i) = \lambda(e_j)$ and $i = j$. Furthermore, a pair of traces (or subsequences of traces) $\sigma = \langle e_1\ e_2 \ldots e_m \rangle, \sigma' = \langle e'_1\ e'_2 \ldots e'_n \rangle$ are order equivalent, denoted as $\sigma \sim_{order} \sigma'$, if $n = |\sigma| = |\sigma'|$ and for every $1 \leq i \leq n$ then $e_i \sim e'_i$. Intuitively, a pair of sequences are order equivalent if they contain equivalent events and the order among them is the same. Let $\widehat{\sigma}$ be the set representation of a sequence σ, where the order between the events is omitted. Then a pair of set representations of σ, σ' are equivalent, denoted as $\sigma \sim_{set} \sigma'$, if they contain equivalent events, i.e., $\forall e_i \in \widehat{\sigma} \exists e'_j \in \widehat{\sigma'} : e_i \sim e'_j$ and vice-versa, $\forall e'_j \in \widehat{\sigma'} \exists e_i \in \widehat{\sigma} : e_i \sim e'_j$.

Every event in a trace has a past and a future. The past of an event e_i, denoted as $\lfloor e_i \rfloor$ in $\sigma = \langle e_1\ e_2 \ldots e_n \rangle$, $1 < i < n$, is the prefix subsequence $\sigma_{[1 \ldots i-1]}$, while the future of e_i in σ, denoted as $\lceil e_i \rceil$, is the suffix subsequence $\sigma_{[i+1,n]}$. The prefix subsequence of the event e_1 is the empty sequence ϵ. For instance, given the trace $\langle i_1\ b_1\ c_1\ d_1\ o_1 \rangle$, the past of b_1 is $\lfloor b_1 \rfloor = \langle i_1 \rangle$ and its future is $\lceil b_1 \rceil = \langle c_1\ d_1\ o_1 \rangle$.

A trace describes the evolution of an execution of a system by means of its prefix subsequences and their extensions. Thus, a trace can be represented as a transition graph where every event is a transition between a pair of execution states. Formally, a transition graph is the tuple $\langle V, v_i, W, E, T \rangle$, where V is the set of states, v_i is the initial state, W is the set of final states, E is the set of events, and T is a transition relation. The next definition suggests how to construct a transition graph from a trace.

Definition 3.3 (*Transition graph of a trace*). Let $\sigma = \langle e_1\ e_2\ \ldots\ e_n \rangle$ be a trace. The transition graph representing σ is defined as $\langle V, \varnothing, \{\widehat{\sigma}\}, E, T \rangle$, where

$$\begin{aligned} V &= \{\widehat{\sigma'} \mid \sigma' \in \phi(\sigma)\} \\ E &= \widehat{\sigma} \\ T &= \{(v, e_i, v \oplus e_i) \mid e_i \in E \ \wedge\ v = \widehat{\lfloor e_i \rfloor}\} \end{aligned}$$

Consider the log $\mathcal{L} = \{\langle i_1\ b_1\ c_1\ d_1\ o_1 \rangle, \langle i_1\ a_1\ c_1\ d_1\ f_1\ o_1 \rangle, \langle i_1\ a_1\ d_1\ c_1\ f_1\ o_1 \rangle\}$, the transition graphs representing the traces in $\mathcal{L}$ are displayed in Fig. 3. Observe that different traces can represent interleavings of the same execution and thus similar execution states. For instance, the two transition graphs at the bottom of Fig. 3 can be seen as interleavings of an execution where c_1 and d_1 are concurrent, which would imply that the states $\{\{i_1\}, \{i_1, a_1\}, \{i_1, a_1, c_1, d_1\}, \{i_1, a_1, c_1, d_1, f_1\}, \{i_1, a_1, c_1, d_1, f_1, o_1\}\}$ represent the execution of the same events and thus are equivalent. Indeed by treating these states as equivalent, the concurrency relation between c_1 and d_1 would appear

[2]Generally speaking, an event log is a multiset of traces, however we focus simply on the ordering of events and disregard the information about the number of times each trace occurs in the log. Furthermore, for simplicity, we assume that every trace is a complete execution and the log is noise free.

as a diamond in the transition graph (cf. Fig. 4) denoting the possible (interleaved) concurrent execution of such events.

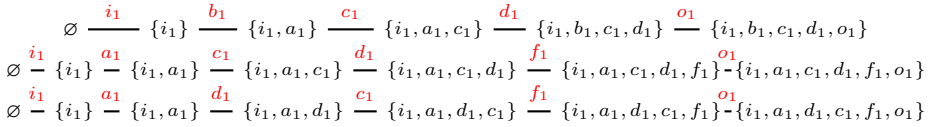


Fig. 3: Transition graphs representing three different traces.

We now turn our attention to the definition of an equivalence relation between states of transition graphs. As hinted previously, the equivalence relation is used to collapse sets of "similar" states, which can introduce some generalization and, by the same token, discover patterns reflecting the concurrent execution of events. The equivalence relation between states defined in this section is grounded on the most primitive interpretation of the (interleaved) concurrent execution of a pair of events. Specifically, staring from an execution state, a pair of concurrent events can occur in any order and lead to the same execution state. The latter is the essence of the results presented in [Nielsen et al. 1981], where the authors show that a transition graph-like representation (*domain of configurations*) can represent the true-concurrency semantics of a system, where the concurrent execution of a pair of events is manifested as diamond-like shapes.

The main idea of the equivalence relation presented below is to construct a transition graph where every state is associated to a unique set of events: the events that have occurred before the state is reached. Each transition is labeled by an event and connects a pair of states, such that the set of events in the *source* state is a strict subset of the set of events of the *target* state. We distinguish two special types of nodes in a transition graph, a unique initial state $\varnothing$, and at least one final state (state with no outgoing transitions), which represent the executions of a process.

Fig. 4 shows an example of the type of transition graph that we seek to extract from a log. The graph represents two executions: $\{i_1, a_1, c_1, d_1, f_1, o_1\}$ and $\{i_1, b_1, c_1, d_1, o_1\}$. In the graphical representation, the sets of events denote states and the events associated to the transitions are displayed aside. Note that in the case of $\{i_1, a_1, c_1, d_1, f_1, o_1\}$, there is a diamond representing the possible concurrent execution of events c_1 and d_1 (cf. gray nodes in Fig. 4). These diamonds define the *scopes* where pairs of events are executed concurrently, and thus give place to a notion of local concurrency relations. The bottom and top states of a diamond are referred to as start and end of the scope, and they represent the states where no concurrent event has been executed and the state where all the concurrent events have taken place, respectively. For example, in Fig. 4, the states $\{i_1, a_1\}$ and $\{i_1, a_1, c_1, d_1\}$ define the start and end of the scope, respectively, where c_1 and d_1 are concurrent. Note that the concurrency relation between c_1 and d_1 does not hold for the other occurrences on the right hand side.

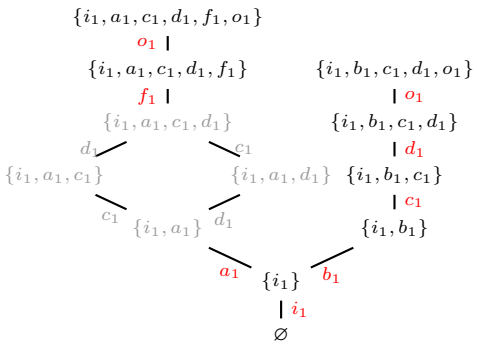


Fig. 4: Transition graph of the log $\mathcal{L} = \{\langle i_1\ b_1\ c_1\ d_1\ o_1\rangle, \langle i_1\ a_1\ c_1\ d_1\ f_1\ o_1\rangle, \langle i_1\ a_1\ d_1\ c_1\ f_1\ o_1\rangle\}$

The equivalence relation defined below deems a pair of states as equivalent if they represent the occurrence of equivalent events and, either they were executed in the same order (they are essentially the same activity occurrences) or the same set of events (in the same order) can occur from both states.

Definition 3.4 (*State equivalence*). Given a pair of traces σ and σ' and their corresponding transitions graphs $G = \langle V, v_i, \{\widehat{\sigma}\}, E, T\rangle$ and $G' = \langle V', v'_i, \{\widehat{\sigma'}\}, E', T'\rangle$, respectively. Let $\sigma_1 = \sigma_{[1\ldots m]}$ and $\sigma'_1 = \sigma'_{[1\ldots m]}$ be a pair of subsequences, such that $v = \widehat{\sigma_1}$ and $v' = \widehat{\sigma'_1}$ are the corresponding states in the graphs. The states v, v' are equivalent — shorthanded as $v \equiv v'$— if $\sigma_1 \sim_{set} \sigma'_1$ and either of the following hold: (i) $\sigma_1 = \sigma'_1 = \epsilon$, (ii) $\sigma_1 \sim_{order} \sigma'_1$, or (iii) $n = |\sigma| = |\sigma'| \ \wedge \ (m = n \ \vee \ \sigma_{[m+1,n]} \sim_{order} \sigma'_{[m+1,n]})$.

The equivalence class of a state v is defined as $\langle v\rangle_{\equiv} = \{v' \mid v \equiv v'\}$.

The equivalence relation between states gives place to an equivalence relation between events. Consider a pair of graphs G, G', and a pair of transitions (v'_1, e_i, v''_1) in G and (v'_2, e_j, v''_2) in G'. Then $e_i \equiv e_j$ iff $v'_1 \equiv v'_2$ and $v''_1 \equiv v''_2$. The equivalence class of an event e_i is denoted as $\langle e_i\rangle_{\equiv} = \{e_j \mid e_i \equiv e_j\}$. For instance, consider the two traces at the bottom of Fig. 3, by Definition 3.4, the states $\varnothing$ are equivalent by rule (i); states representing both $\{i_1\}$ and $\{i_1, a_1\}$ are equivalent by rule (ii); and states representing $\{i_1, a_1, c_1, d_1\}, \{i_1, a_1, c_1, d_1, f_1\}$ and $\{i_1, a_1, c_1, d_1, f_1, o_1\}$ are equivalent by rule (iii). The equivalent events are those with labels i_1, a_1, f_1 and o_1.

Given an equivalence notion $\equiv$, e.g., the one presented in Def. 3.4, the construction of a transition graph from an event log is presented next.

Definition 3.5. Let $\mathcal{L}$ be an event log. The transition graph of $\mathcal{L}$ is defined as $G = \langle V, v_i, W, E, T\rangle$, where

$$
\begin{array}{lcl}
V & = & \{\langle v\rangle_{\equiv} \mid v = \widehat{\sigma'} \ \wedge \ \sigma' \in \phi(\sigma)\} \\
v_i & = & \varnothing \\
W & = & \{\langle \widehat{\sigma}\rangle_{\equiv}\} \\
E & = & \{\langle e_i\rangle_{\equiv} \mid 1 \leq i \leq |\sigma|\} \\
T & = & \{(\langle v_1\rangle_{\equiv}, \langle e_i\rangle_{\equiv}, \langle v_2\rangle_{\equiv}) \mid \exists \sigma', \sigma'' \in \phi(\sigma), e' \in \langle e_i\rangle_{\equiv} \ : \ v_1 = \widehat{\sigma'} \ \wedge \ v_2 = \widehat{\sigma''} \ \wedge \\
 & & v_2 = v_1 \cup \{e'\}\}
\end{array}
$$

for all traces σ in $\mathcal{L}$.

The transition graph representing the event log $\mathcal{L} = \{\langle i_1\ b_1\ c_1\ d_1\ o_1\rangle, \langle i_1\ a_1\ c_1\ d_1\ f_1\ o_1\rangle, \langle i_1\ a_1\ d_1\ c_1\ f_1\ o_1\rangle\}$ constructed with the equivalence in Def. 3.4 is that of Fig. 4.

3.2. Discovering scopes of concurrency

Once the transition graph is constructed, the second step consists of turning an existing global concurrency oracle into a scoped (local) one. Specifically, the aim is to discover parts of a transition graph where concurrency relations between events are likely to hold. We refer to this parts of the transition graph as scopes and denote them as $\mathcal{S}$. The approach requires a (global) concurrency oracle $\mathcal{C}$, which computes a set of relations from a given set of traces, and a validation function $\mathcal{F} : \mathcal{S} \times (E \times E) \rightarrow \{true, false\}$ that, given a concurrency relation in a scope, retrieves a boolean value representing the outcome of the validation.

In a transition graph, a *path* between a pair of states v_1 and v_2, shorthanded as $\pi(v_1, v_2)$, is a sequence of transitions $\langle(v_1, e_1, v_a)\ (v_a, e_2, v_b)\ \ldots (v_x, e_x, v_2)\rangle$. Note that $\pi(v_1, v_2)|_E$ is used to refer only to the events in a path $\pi(v_1, v_2) = \langle(v_1, e_1, v_a)\ \ (v_a, e_2, v_b)\ \ \ldots (v_x, e_x, v_2)\rangle$, i.e., $\pi(v_1, v_2)|_E = \langle e_1\ \ e_2 \ldots e_x\rangle$; while, $\pi(v_1, v_2)|_{\lambda(E)} = \langle\lambda(e_1)\ \lambda(e_2) \ldots \lambda(e_x)\rangle$ is used to refer to the event types in π. A state v_x is in $\pi(v_1, v_2)$, denoted as $v_x \in \pi(v_1, v_2)$, if there is a transition (v_i, e_k, v_j) in $\pi(v_1, v_2)$,

such that $v_x = v_i$ or $v_x = v_j$. The set of all distinct paths from v_1 to v_2 is represented as $\Pi(v_1, v_2)$, while the distinct sequences of events and event types induced by such paths are denoted by $\Pi(v_1, v_2)|_E$ and $\Pi(v_1, v_2)|_{\lambda(E)}$, respectively. By the abuse of notation, let $e_x \in \Pi(v_1, v_2)$ denote the existence of a path between v_1 and v_2 that includes e_x and, similarly for transitions, let $(v_a, e_i, v_b) \in \Pi(v_1, v_2)$ denote the existence of a path including the transition (v_a, e_i, v_b). Then, a *scope* $\mathcal{S}$ is a pair of start v_s and end v_e states, such that there is at least one path from v_s to v_e in the graph, i.e., $|\Pi(v_s, v_e)| > 1$.

Given a scope $\mathcal{S}$, first we compute the paths in the scope and give them as input to the global concurrency oracle $\mathcal{C}$. Then, the concurrency relations retrieved by $\mathcal{C}$ are evaluated by $\mathcal{F}$; such that only those evaluated to true are set as valid in $\mathcal{S}$. If a concurrency relation between a pair of events (e_1, e_2) holds within $\mathcal{S}$, then $\mathcal{S}$ is called a *concurrency scope* for (e_1, e_2).

Definition 3.6 (*Concurrency scope*). Let $G = \langle V, i, W, E, T\rangle$ be a transition graph, $\lambda : E \to \Lambda$ be a labelling function, $\mathcal{C}$ be a concurrency oracle, and $\mathcal{F}$ be a validation function. A concurrency scope $\mathcal{S}$ is the tuple $\langle v_s, v_e, (e_1, e_2)\rangle$, where $v_s \in E$ is the start state, $v_e \in E$ is the end state and $(e_1, e_2) \in E \times E$ is a pair of events, such that $(\lambda(e_1), \lambda(e_2)) \in \mathcal{C}(\Pi(v_s, v_e)|_{\lambda(E)})$, $e_1, e_2 \in \Pi(v_s, v_e)$ and $\mathcal{F}(\mathcal{S}, (\lambda(e_1), \lambda(e_2))) = true$.

Intuitively, a concurrency scope for a pair of events (e_1, e_2) is valid if their types are deemed concurrent by the global concurrency oracle, they appear at least in one path in the scope, and the validation function asserts such relation. We turn our attention to the computation of the scopes where we rely on well known concepts of graph theory, dominator and post-dominator relations. Intuitively, in any directed graph, a vertex a is the dominator of a vertex b if every path from an initial node i to b contains a. For directed graphs with a final vertex f, we say that a vertex z is the post-dominator of a vertex y if all paths from y to f contain z. Both, dominator and post-dominator relations are reflexive and transitive, and their transitive reduction are rooted trees referred to as dominator tree $\mathcal{T}_{dom}$ and post-dominator tree $\mathcal{T}_{post}$, respectively.

As shown in the example displayed in Figure 4, the concurrency relations between a pair of events can hold only in certain executions. Thus, our approach will decompose a transition graph with many final states (each of them representing an execution) into subgraphs from the initial state $\varnothing$ to each of the final states, and then compute the concurrency scopes for each of those subgraphs.

The algorithm for the computation of the concurrency scopes is displayed in Algorithm 1. The algorithm receives as input a transition graph, a global concurrency oracle and the validation function. Then it starts by iterating over the subgraphs G' from the initial state to one of the final states v in G (Line 3). For each subgraph G', the dominator and post-dominator tree are constructed (Lines 4 and 5). In a postorder manner in the dominator tree, the start of a scope is selected v_e (Line 6), while the end of the scope is the parent of v_e in the post-dominator tree (Line 8). In Line 10, each of the concurrency relations retrieved by the local concurrency oracle is checked in the scope (v_s, v_e). If the validation function asserts the concurrency relation between a pair of event types (a, b) in a scope (v_s, v_e), then it is added as a concurrency scope associated to a final state v (Line 19). The algorithm considers two operations for expanding a concurrency scope whenever a concurrency relation holds (Line 20-22), or restricting the scope when the validation function fails and smaller concurrency scopes could be detected (Line 25-28). The output of the algorithm is a set of tuples $(v, \mathcal{S})$, where v is a set of events representing a final state, and $\mathcal{S}$ is a concurrency scope. Observe that a single final state v can be associated to many concurrency scopes. Thus, given an event log, the concurrency scopes for a trace σ are those in $\{(\widehat{\sigma}, \mathcal{S})\}$.

Next, we present an example of a concurrency oracle and a validation function that could be plugged into the algorithm.

Algorithm 1: Computing concurrency scopes

Algorithm
  Input: Transition graph $G = \langle V, \varnothing, W, E, T \rangle$, global concurrency oracle $\mathcal{C}$ and validation function $\mathcal{F}$.
  Output: Concurrency scopes $\mathcal{O} = \{(v, \mathcal{S}) \mid \mathcal{S}$ is a concurrency scope, and $v \in W\}$
  $\mathcal{O} \leftarrow \varnothing$
  for $G' = \langle V', \varnothing, v, E', T' \rangle$ *such that* $v \in W$ **do**
    $\mathcal{T}_{dom} \leftarrow$ dominator tree of G'
    $\mathcal{T}_{post} \leftarrow$ post-dominator of G'
    for v_s *in* $\mathcal{T}_{dom}$ *in postorder* **do**
      if $v_s \neq v$ *and* v_s *has a parent in* $\mathcal{T}_{post}$ **then**
        $v_e \leftarrow$ parent of v_s in $\mathcal{T}_{post}$
        foreach $(a, b) \in \mathcal{C}(\Pi(v_s, v_e)|_{\lambda(E)})$ **do**
          `computeScope`$(v, (v_s, v_e), (a, b))$
        end
      end
    end
  end
  return $\mathcal{O}$
Procedure `computeScope`$(v, (v_s, v_e), (a, b))$
  if $\mathcal{F}((v_s, v_e), (a, b))$ **then**
    $\mathcal{O} \cup \{(v, \mathcal{S})\}$, such that
      $\mathcal{S} = \langle v_s, v_e, (a_i, b_j) \rangle$ and $a_i, b_j \in \Pi(v_s, v_e)$ ▷ Add as concurrency oracle
    if v_e *has a parent in* $\mathcal{T}_{post}$ **then**
      $v_e \leftarrow$ parent of v_e in $\mathcal{T}_{post}$ ▷ Expand
      `computeScope`$((v_s, v_e), (a, b))$
    end
  else
    if v_e *has a child in* $\mathcal{T}_{post}$ **then**
      $v_e \leftarrow$ child of v_e in $\mathcal{T}_{post}$ such that (v_s, v_e) is a scope ▷ Restrict
      `computeScope`$((v_s, v_e), (a, b))$
    end
  end

3.2.1. Global (baseline) concurrency. We rely on existing concurrency oracles for the computation of the concurrency relations between pairs of events, e.g. [van der Aalst et al. 2004; Mokhov and Carmona 2015; Cook and Wolf 1998], and assume that no two events with the same label can be executed concurrently. As a baseline we use the concurrency relation introduced in [van der Aalst et al. 2004], herein referred to as α concurrency. Intuitively, a pair of labels a, b are concurrent if a is sometimes observed immediately after b and vice-versa. The definition of α concurrency is given next.

Definition 3.7 (α *concurrency [van der Aalst et al. 2004]*). Let σ be an event trace. A pair of tasks with labels $a, b \in \mathcal{L}$ are said to be in α *directly precedes relation*, denoted $a \prec b$, iff there exists a trace $\sigma = \langle e_1\ e_2 \dots e_n \rangle$ in L, such that $a = \lambda(e_i)$ and $b = \lambda(e_{i+1})$, $1 \leq i \leq n$. A pair of tasks a, b are α *concurrent*, denoted $a \parallel b$, iff $a \prec b \wedge b \prec a$.

Oftentimes the concurrency relations computed by the concurrency oracle, and in particular by the α concurrency, can be spurious. For instance, consider the trace $\langle a_1\ b_1\ c_1\ d_1\ b_2\ a_2 \rangle$, where the α concurrency would deem events with labels a, b as concurrent. Thus, the validation function is used to refine the results of the concurrency oracle and filter out spurious concurrency relations.

3.2.2. Validation of concurrency relations. As an example, we define a validation function based on the proportion of events deemed as concurrent that can be executed from the same states in the scope.

Definition 3.8 (*Validation function*). Let $co(a, b)|_{(v_s, v_e)} = \{v \in \Pi(v_s, v_e) \mid (v, a_i, v'), (v, b_j, v'') \in \Pi(v_s, v_e)\}$ be the set of states within the

paths $\Pi(v_s, v_e)$ where events with labels a and b can occur. Additionally, let $\#(a) = \{(v, a_i, v') \in \Pi(v_s, v_e)\}$ be the set of transitions associated to the occurrence of an event with label a in $\Pi(v_s, v_e)$. Then, the occurrence of a, b w.r.t. to a is $f(a) = \frac{|co(a,b)|_{(v_s,v_e)}|}{|\#(a)|}$ and, similarly w.r.t. b, $f(b) = \frac{|co(a,b)|_{(v_s,v_e)}|}{|\#(b)|}$. The event types a, b are concurrent in (v_s, v_e) iff (1) $f(a) > tOccurrence$, (2) $f(b) > tOccurrence$, and (3) $abs(f(a) - f(b)) < tBalance$, for some thresholds $tOccurrence$ and $tBalance$.

Intuitively, the validation function checks that the number of events with labels a, b can often be executed from the same state w.r.t. to their total number of occurrences (i.e., the proportion is higher than a given threshold $tOccurrence$) and that the proportions between those events are similar enough (i.e., not bigger than $tBalance$).

3.2.3. Complexity analysis. The worst-case time complexity of our algorithm is dominated by the complexity of constructing the transition graph out of an event log, and the complexity of computing the scopes and the traces within them. The complexity of these steps is polynomial. Given a log with n number of events, the construction of the transition graph is done in $O(\frac{n(n+1)}{2})$. This is because for every event e_i in the log, $1 \leq i \leq n$, it is necessary to spot the state in the transition graph that reflects the execution of e_i w.r.t. to the given equivalence relation. If there is no equivalent state then a new state is added. Thus, the number of comparison operations for finding equivalent states increases, at most, together with the number of events analysed.

The case for the computation of the scopes and the traces within a scope is as follows. Given a transition graph with V states, T transitions and a unique final state, the computation of the dominator and post-dominator trees can be done in $O((|T| + |V|)log(|T| + |V|))$ with the Lengauer-Tarjan algorithm. Independently of the traversals of the dominator and post-dominator trees, there are at most $|V|^2$ scopes (all possible combinations of start-end states). Then, given a scope $\mathcal{S}$ with E' events and V' states, there can be up to a factorial number of paths (i.e., all interleavings of the concurrent execution of the events E') in $\mathcal{S}$. However, using dynamic programming techniques and given that the transition graphs are directed and acyclic, the computation of the paths can be done in $O(|V'| + |T'|)$, where T' is the number of transitions in the scope.

The next section presents the evaluation with a set of synthetic logs and uses the concurrency oracle and validation function presented above.

4. EVALUATION OF ACCURACY AND TIME PERFORMANCE

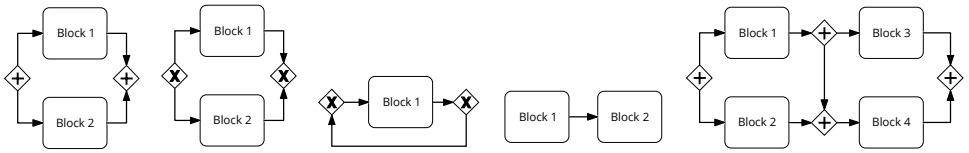
We implemented our local concurrency oracle in a Java tool called *ProLoCon*[3] and used this tool to evaluate the approach's accuracy and time performance. The tool takes as input an event log in MXML or XES format, a concurrency oracle (currently the α oracle [van der Aalst et al. 2004] and the oracle in [Mokhov and Carmona 2015] are supported), as well as the values of the thresholds *tOccurrence* and *tBalance* for the validation function.

4.1. Datasets generation

In order to evaluate the accuracy of our approach, we defined a gold standard by generating a set of synthetic process models capturing a wide combination of control-flow constructs. These single-entry single-exist (SESE) models were obtained by randomly composing the following SESE fragments: AND, XOR, Loops, Sequences and Z-blocks (see Figure 5 for the BPMN notation of each fragment). The models were generated as trees of height two, where every leaf is an activity and every internal node is a fragment either containing nested fragments or atomic activities. The leaves of the trees are activities randomly chosen and duplicate activities are allowed as long as they

[3]Available at `http://apromore.org/platform/tools`

do not introduce auto-concurrency, i.e., pairs of activities with the same label cannot belong to the same parallel block. This led to a total of 82 models, ranging from a minimum size of 10 nodes to a maximum size of 20 nodes (avg. = 15.5 nodes). Out of these models, ten are cyclic and all include at least one pair of concurrent events.



(a) AND (b) XOR (c) Loop (d) Sequence (e) Z block

Fig. 5: Fragments used for the generation of synthetic process models.

For generating the logs from these synthetic models we used the ProM plugin *Generate Event Log from Petri Net* [Vanden Broucke et al. 2012].[4] The obtained logs vary from a minimum of 4 to a maximum of 300 traces (avg. = 24 traces), with a total number of events ranging from 24 to 2,400 (avg. = 173 events).

4.2. Setup

Using the synthetic models as a gold standard, we computed the *F-score* between the concurrency relations identified in the log and those extracted from the respective model, for each model-log pair in our dataset. To do so, we relied on an existing technique [García-Bañuelos et al. 2015] for the comparison of Prime Event Structures (PESs). The evaluation framework is shown in Fig. 6.

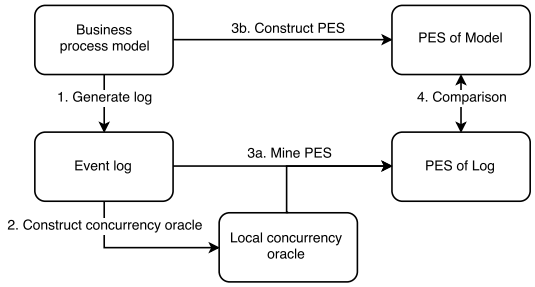


Fig. 6: Evaluation framework (artificial logs).

PESs [Nielsen et al. 1981] have been suggested as a suitable formalism for a unified behavioral representation of a log and a process model in the context of process mining [Dumas and García-Bañuelos 2015]. Intuitively, a PES is a model of concurrency describing the behavior of a system by means of events and three different binary relations between them, namely concurrency, conflict and causality. The execution semantics of a PES is described by means of *configurations*, sets of events capturing the execution states, and *extensions*, events that can occur at a given configuration. Then, if two events are concurrent, they are extensions of at least one configuration.

[4]Parameters of the simulation: complete generation; min./max. traces to add for each generated sequence: 1; max. times marking seen: 2; only include traces that reach end state; only include traces without remaining tokens.

To construct the PES from the model and that from the log, we used the method proposed in [García-Bañuelos et al. 2015]. This method extends the classical notion of PES [Nielsen et al. 1981] to represent the behavior of both acyclic and cyclic process models, as well as the behavior encoded in a log. In the latter case, it requires as input any concurrency oracle for transforming the traces in the log into partial ordered sets of events which are then used to build the PES.

According to [García-Bañuelos et al. 2015], given a pair of PESs P_{log}, P_{model} from a log and a model, respectively, a pair of configurations s_1 of P_{log} and s_2 of P_{model} are equivalent if they represent the occurrence of equivalent events. Then, let TP (true positives) be the set of equivalent configurations s_1, s_2 where for any pair of concurrent events a_i, b_j which are extensions of s_1 in P_{log}, there is a pair of concurrent events a_k, b_l extending s_2 in P_{model}, such that $a_k \sim a_i$ and $b_j \sim b_l$. Let FP (false positives) be the set of equivalent configurations s_1, s_2, where there is a pair of concurrent events a_i, b_j that are extensions of s_1 in P_{log} and for which there is no equivalent (concurrent) events extending s_2. Finally, let FN (false negatives) be the set of equivalent configurations s_1, s_2, such that there is a pair of concurrent events a_k, b_l extending s_2 in P_{model} and for which there is no equivalent (concurrent) events extending s_1.

Having defined the sets of true positives, false positives and false negatives, we can compute Precision and Recall, and in turn the F-score.

4.3. Results

Before measuring the accuracy of our approach, we did a sensitivity test on the thresholds *tOccurrence* and *tBalance*. Figure 7.a shows how the F-score varies according to different combinations of *tOccurrence* and *tBalance*. We observe that the F-score plateaus at a value of 0.977 with *tOccurrence*=0.4 and *tBalance*=0.2. The same result was obtained with a *tOccurrence*=0.5. In the case of a *tOccurrence* grater than 0.5, the validation function resulted too strict and no concurrency relation could be detected, leading to a very low F-score. Hence, we selected *tOccurrence*=0.4 and *tBalance*=0.2 for our accuracy evaluation.

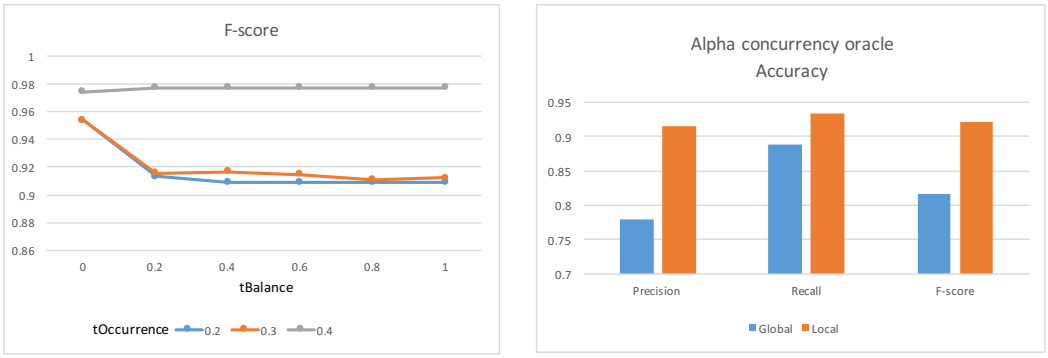


Fig. 7: (a) Sensitivity test for the parameters *tOccurrence* and *tBalance*. (b) Average precision, recall and F-score of our oracle against the α oracle.

The other parameters used for the evaluation are the state equivalence of Definition 3.4, and the α concurrency as the global baseline concurrency oracle (Def. 3.7). We also used the technique in [Mokhov and Carmona 2015] as a global baseline oracle. However, the concurrency relations identified were exactly the same as those retrieved by the α concurrency.

Figure 7.b reports the average results of the accuracy evaluation across all 82 model-log pairs. The bar diagram shows a sensible increment in F-score (from 0.82 to 0.92), mostly determined by the increase in precision when using our local concurrency oracle

instead of the global one (from 0.78 to 0.92). The lower precision of the α oracle is due to its over-generalization, given that local concurrency relations are identified as global relations. The lower recall of the α oracle is due to the assumption that the log is complete w.r.t. the direct follows relation, meaning that all possible such relations are expected to be present in the log, for the oracle to accurately detect the α-relations. However, this assumption hardly holds in real-life datasets, hence we did not enforce log completeness when generating the logs for our experiments.

There were two problematic constructs where the local concurrency oracle failed to accurately determine the scope of a pair of concurrency relations. One construct is when there are two AND blocks containing the same tasks and following each other. For example, given a log $\{\langle a\ b\ a\ b\rangle, \langle b\ a\ a\ b\rangle, \langle a\ b\ a\ b\rangle, \langle a\ b\ b\ a\rangle\}$, the local concurrency oracle identifies as concurrent every pair of a and b from the beginning to the end of every trace, and thus fails to identify that the second occurrence of a and b depends on the first occurrence. The other problematic construct is a sequence of a loop of an activity a, followed by a concurrent block of activities a and b. In this case, the concurrency oracle identifies the event in the loop as concurrent with b. These cases are however also misclassified by the global baseline oracle.

Table I reports the statistics on the execution time (in milliseconds) for the computation of the concurrency relations using the global α oracle and its local counterpart. Even though the execution times for the computation of our local concurrency oracle are sensibly higher than those of the global one, the overall time taken is still quite low, in the order of milliseconds (average of 6.9 ms).

	Global (ms)	**Local (ms)**
Max	0.788	172.064
Min	0.006	0.392
Avg	0.060	6.867

Table I: Execution times of our oracle against the α oracle.

5. EVALUATION OF GENERALIZATION EFFECTS

As a second experiment, we evaluated the effects of using a local concurrency oracle on the generalization of the process behavior captured in the event log. First, we compared the generalization introduced by the local concurrency oracle with that introduced by the global concurrency oracle. Second, in order to investigate the possible over-generalization inherent to existing process discovery algorithms, we compared the generalization introduced by the local concurrency oracle with that of the Inductive Miner [Leemans et al. 2013], which is a state-of-the-art automated discovery technique. Specifically, given that the Inductive Miner can generate a Petri net out of an event log, which is able to replay every trace in the log, we used such model as an oracle (referred to as *"inductive"* oracle hereinafter) that transforms a trace into a partial order (process induced by replaying the trace on the net).

For this second experiment we used seven real-life logs. The first log is that distributed with the BPI Challenge (BPIC) 2012; it captures executions of a personal loan origination process at a Dutch financial institute.[5] The second log captures executions of an IT service desk process at an Italian IT Vendor, for handling both service requests and incidents. The third log captures executions of a business process for plan lodgement and document registration in two Australian states, as recorded by a land development company, whose model is depicted in Figure 1. The fourth, fifth and sixth logs capture executions of a process for handling motor glass claims at an Australian insurance company. Finally, the last event log is extracted from an information system

[5] doi:10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f

Log	Events	Event types	Traces	Distinct traces	Avg trace length	TG	Alpha Conc. relations			Inductive Conc. relations		
							TL	FC	c_{og}	TL	FC	c_{og}
BPIC 2012	262,200	23	13,087	4,336	42	0	44	35	1	15	0	1
IT Vendor	75,353	9	12,720	1,026	13	0	8	1	1	7	0	1
Land dev.	18,240	14	1,440	36	12.6	5	1	1	0.28	1	1	0.29
Insurance 1	19,544	12	4,954	74	4.39	6	11	1	0.67	11	0	0.65
Insurance 2	7,606	11	1,853	38	4.63	6	2	0	0.25	2	0	0.25
Insurance 3	52,361	12	13,302	102	4.50	5	15	1	0.76	15	1	0.76
Traffic fines	561,470	11	150,370	231	8.18	0	26	2	1	26	0	1

Table II: Statistics on real-life logs and their concurrency relations.

for managing road traffic fines in Italy.[6] The characteristics of these logs are reported in the first part of Table II.

To measure the effects of generalization, we first transformed each trace of each real-life log into a partial order run using the local, global and inductive oracles. Next, we measured the number of *true global* (TG) concurrency relations, i.e. a concurrency relation between events a_1, b_1 is considered true global if there is a scope computed by the local oracle for every occurrence of the event types a, b; the number of *true local* (TL) relations (or *false global*), i.e. those relations that were identified by the global (inductive) oracle, for which our oracle found a local scope; and the number of *false concurrent* (FC) relations, i.e. those pairs of events which the global (inductive) oracle identified as being concurrent, but were not found to be concurrent at all by our local oracle. As an example, with reference to Fig. 1, the concurrency relation between "Update register" and "Update DCDB" is an example of false concurrency, since these two event types are actually never concurrent, while the concurrency relation between "Approve plan" and "Update register" is an example of true local concurrency.

Using these measures, we then computed the *concurrency over-generalization ratio* as the ratio between the number of false global and false concurrent relations, and the total number of concurrency relations found by the global (inductive) oracle, i.e. $c_{og} = \frac{TL+FC}{TG+TL+FC}$. In essence, this formula measures how much the global oracle over-generalizes the behavior captured in the log, either by identifying a local concurrency relation as being global, or by identifying a pair of events as being (globally) concurrent when they are not. The results are reported in the second part of Table II for both the global and the inductive oracles.

As we can observe, the global and inductive oracles have a concurrency over-generalization ratio ranging from 25% in the case of one of the insurance company logs, to 100% in the case of the BPIC 2012, IT vendor and Traffic fines logs. The high value obtained in these latter logs suggests that many such concurrent relations are indeed authentically local. This observation is supported by the average number of repeated events per trace in these three logs, i.e. the ratio between the number of events in a trace and the number of event types in that trace, averaged across all traces. This ratio is 1.3 in the Traffic files log (30% of events are repeats of other events) and as high as 4.3 in the IT Vendor log and 4.4 in the BPIC log (i.e. an event type appears on average four times in a trace). However, some of these local relations may actually be due to the incompleteness of the log. In this case, the validation function of our oracle may turn out to be too strict because not all the interleavings of a given concurrency relation are captured in the log, leading to a reduced scope of the relation, hence to local concurrency.

Figure 8 shows an example of a trace extracted from the Traffic fines log, that resulted in two non-isomorphic partial order runs, after relaxing the order relations of the events in the trace according to the local oracle and to the global oracle. As we can see, the run obtained with the global oracle identifies the "Payment" event as being

[6] doi:10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5

concurrent with all other events. This is a clear case of concurrency over-generalization because logically the payment for a fine can only be done after the fine has actually been sent to the citizen. Similarly, event "Receive result appeal from prefecture" is concurrent to "Insert date appeal to prefecture", which again is logically not possible. This is due to the repetition of such events within the same traces in the log. On the other hand, our local oracle identifies the correct order of these events, by placing "Payment" at the end of the run, and "Insert fine notification" and "Add penalty" in a local concurrency relation with "Inser date appeal to prefecture", with the latter event always preceding "Receive result appeal from prefecture".

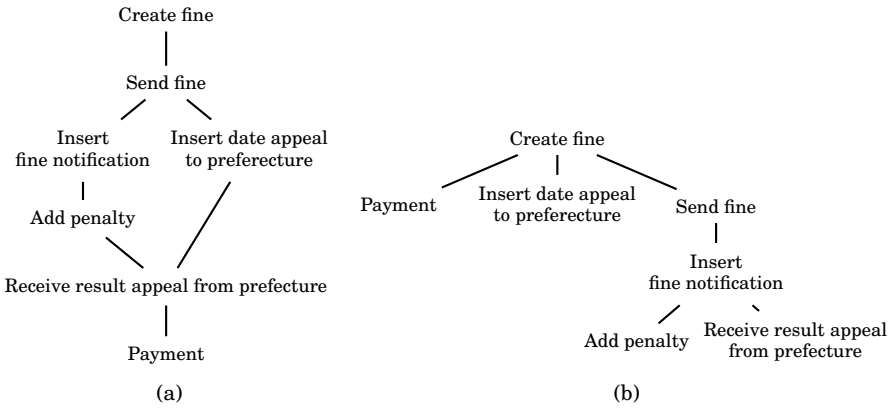


Fig. 8: Traffic fines log: partial order runs derived from the same trace using the local oracle (a) and the global oracle (b).

6. APPLICATION TO PROCESS DRIFT DETECTION

In this section, we present the results of an experiment conducted to study the impact of using our local concurrency oracle instead of the global one for process drift detection.

Early detection of business process changes based on event logs, also known as *process drift detection*, enables analysts to identify and act upon changes that may otherwise affect process performance. In [Maaradji et al. 2015], we introduced a drift detection method where the basic idea is to extract the set of completed process traces from two juxtaposed time windows (sliding over a log or a stream of traces), feed the traces within each window into a concurrency oracle to build partial order runs, and perform a statistical test over the observed runs in the two juxtaposed time windows to detect statically significant changes. These changes are a proxy for process drifts. The global α concurrency oracle was used to discover the concurrent relations.

This experiment was performed on an event log recording 2,259 execution traces of a commercial claims handling process at an Australian insurer. The log spans over a period of one year and contains 13,454 events of which twelve are distinct. We employed the drift detection method in [Maaradji et al. 2015] using both the global and the local concurrency oracles.

Figure 9 reports the $P–value$ of consecutive statistical tests based on the two oracles. Overall, the two plots reveal similar behavior. Indeed, two drifts were detected at the same position by the detection method when using either oracle. Nevertheless, the use of the global oracle led to detecting a further drift at trace 386 which was not detected when using the local concurrency oracle.

Upon inspection of the runs underpinning the only drift not detected by the local oracle, we found that the global oracle discovered a concurrent relation in the window

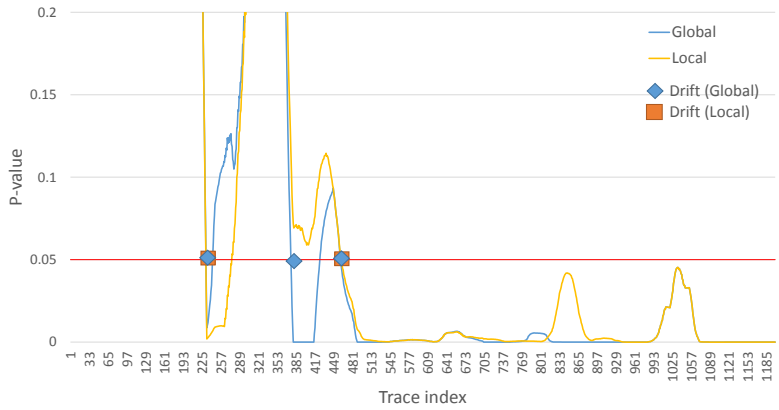


Fig. 9: Plot of the $P–value$ of the statistical test for drift detection, using both the global and local concurrency oracles.

before the drift, but not in the window after the drift. This false concurrency relation was discovered even if one of the interleavings between the two events involved in the relation (from “ReviewApprovePayment” to “ReviewInvoice”) was much less frequent than the other interleaving, as shown in Fig. 10, which plots the relative frequency of the two interleavings over the log.

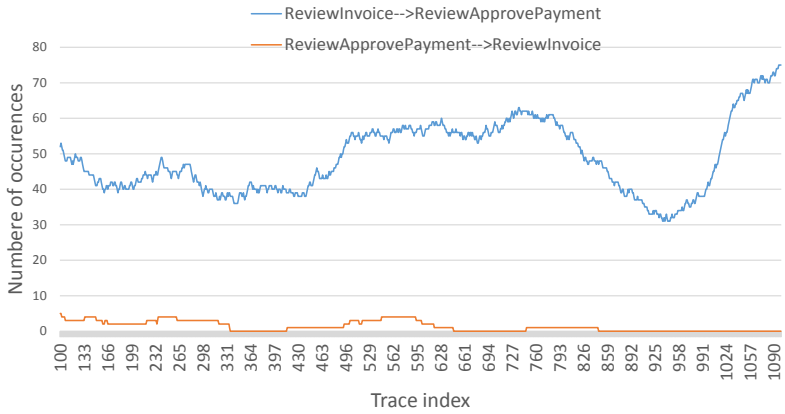


Fig. 10: Plot of the number of occurrences of the interleaved event labels over a sliding window (initialized with 100 traces).

7. CONCLUSION AND FUTURE WORK

This paper presented an approach to turn any algorithm for constructing a global concurrency oracle from an event log into one that constructs local oracles. By scoping the concurrency to a set of states in a state transition graph constructed from the event log, the approach effectively increases the accuracy of the detected set of concurrency relations, while avoiding over-generalization of the process behavior captured in the log. Experimental results have shown that the local concurrency oracles derived from the proposed approach outperform the corresponding global oracles when applied to the task of extracting partially ordered runs from an event log. An application in the

context of business process drift detection has shown the ability of the derived local concurrency oracles to enhance the accuracy of existing process mining methods.

The experimental evaluation suggests that there is room for improvement in the proposed method, particularly the local concurrency oracle fails to find accurate scopes in cases where two blocks of concurrency — including the same event types — precede each other, which leads to detecting bigger scopes than the actual, and when loops and concurrency blocks with common event types precede each other, in which case the event in the loop is detected as concurrent with the event types of the concurrent block. A more extensive evaluation with other (global) concurrency oracles and other parameters for constructing the transition system could inform the development of more robust variants of the proposed method.

Another direction for future work is to explore other applications of the proposed concurrency oracle, for example by combining it with techniques for conformance checking [García-Bañuelos et al. 2015], log delta analysis [van Beest et al. 2015] and automated process discovery [Fahland and van der Aalst 2013], which are based on models of concurrency based on partial orders, and may thus potentially benefit from a finer-grained distinction between causality and concurrency relations.

Acknowledgments. This research is funded by the Australian Research Council (grant DP150103356) and the Estonian Research Council (grant IUT20-55).

REFERENCES

Jonathan E. Cook and Alexander L. Wolf. 1998. Event-based Detection of Concurrency. In *FSE*. ACM, New York, NY, USA.

A. K. A. de Medeiros, B. F. van Dongen, W. M. P. van der Aalst, and A. J. M. M. Weijters. 2004. *Process mining: Extending the α-algorithm to mine short loops*. Technical Report BETA WP 113. Eindhoven University of Technology, Eindhoven.

Claudia Diamantini, Laura Genga, Domenico Potenaa, and Wil van der Aalst. 2016. Building instance graphs for highly variable processes. *Expert Syst. Appl.* 59, 15 (October 2016).

Marlon Dumas and Luciano García-Bañuelos. 2015. Process Mining Reloaded: Event Structures as a Unified Representation of Process Models and Event Logs. In *Proc. of PETRI NETS*. Springer. DOI:http://dx.doi.org/10.1007/978-3-319-19488-2_2

Dirk Fahland and Wil M. P. van der Aalst. 2013. Simplifying discovered process models in a controlled manner. *Inf. Syst.* 38, 4 (2013), 585–605.

Luciano García-Bañuelos, Nick R.T.P. van Beest, Marlon Dumas, and Marcello La Rosa. 2015. *Complete and interpretable conformance checking of business processes*. Technical Report. BPM Center. http://eprints.qut.edu.au/91552/

Sander JJ Leemans, Dirk Fahland, and Wil MP van der Aalst. 2015. Using Life Cycle Information in Process Discovery. In *in press) Business Process Management Workshops*.

Sander J. J. Leemans, Dirk Fahland, and Wil M. P. Aalst. 2013. *Application and Theory of Petri Nets and Concurrency: 34th International Conference, PETRI NETS 2013, Milan, Italy, June 24-28, 2013. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter Discovering Block-Structured Process Models from Event Logs - A Constructive Approach. DOI:http://dx.doi.org/10.1007/978-3-642-38697-8_17

Jiafei Li, Dayou Liu, and Bo Yang. 2007. Process Mining: Extending α-Algorithm to Mine Duplicate Tasks in Process Logs. In *Proc. of APWeb/WAIM 2007 Workshops*. Springer.

Xixi Lu, Dirk Fahland, and Wil M. P. van der Aalst. 2014. Conformance Checking Based on Partially Ordered Event Data. In *Proc. of BPM Workshops*. Springer.

Abderrahmane Maaradji, Marlon Dumas, Marcello La Rosa, and Alireza Ostovar. 2015. Fast and Accurate Business Process Drift Detection. In *Proc. of Business Process Management (Lecture Notes in Computer Science)*, Vol. 9253. Springer, 406–422.

Andrey Mokhov and Josep Carmona. 2015. Event Log Visualisation with Conditional Partial Order Graphs: from Control Flow to Data. In *Proc. of ATAED*.

A. Mokhov and A. Yakovlev. 2010. Conditional Partial Order Graphs: Model, Synthesis, and Application. *IEEE Trans. Comput.* 59, 11 (Nov 2010). DOI:http://dx.doi.org/10.1109/TC.2010.58

Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. 1981. Petri nets, event structures and domains, part I. *Theoretical Computer Science* 13, 1 (1981). DOI:http://dx.doi.org/10.1016/0304-3975(81)90112-2

Hernán Ponce-de León, César Rodríguez, Josep Carmona, Keijo Heljanko, and Stefan Haar. 2015. Unfolding-Based Process Discovery. In *Proc. of ATVA*. Springer.

Nick R. T. P. van Beest, Marlon Dumas, Luciano García-Bañuelos, and Marcello La Rosa. 2015. Log Delta Analysis: Interpretable Differencing of Business Process Event Logs. In *Proc. of BPM*. Springer.

W. M. P. van der Aalst, V. Rubin, H. M. W. Verbeek, B. F. van Dongen, E.Kindler, and C. W. C.W. Günther. 2008. Process mining: a two-step approach to balance between underfitting and overfitting. *Software & Systems Modeling* 9, 1 (2008). DOI:http://dx.doi.org/10.1007/s10270-008-0106-z

Wil M. P. van der Aalst, Ton Weijters, and Laura Maruster. 2004. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Trans. Knowl. Data Eng.* 16, 9 (2004). DOI:http://dx.doi.org/10.1109/TKDE.2004.47

Boudewijn F. van Dongen, Jörg Desel, and Wil M. P. van der Aalst. 2012. Aggregating Causal Runs into Workflow Nets. *Trans. Petri Nets and Other Models of Concurrency* 6 (2012).

Boudewijn F. van Dongen and Wil M. P. van der Aalst. 2004. Multi-phase Process Mining: Building Instance Graphs. In *Proc. of ER*. Springer.

S. Vanden Broucke, J. De Weerdt, J. Vanthienen, and B. Baesens. 2012. *An Improved Process Event Log Artificial Negative Event Generator*. Technical Report. Faculty of Economics and Business, KU Leuven (Belgium).

Lijie Wen, Wil M. P. Aalst, Jianmin Wang, and Jiaguang Sun. 2007. Mining process models with non-free-choice constructs. *Data Mining and Knowledge Discovery* 15, 2 (2007). DOI:http://dx.doi.org/10.1007/s10618-007-0065-y